# CS 611
# Advanced Programming Languages

Andrew Myers
Cornell University

Lecture 27
let- and parametric polymorphism
3 Nov 04

## Polymorphic types

- Type expression may have some unsolved type identifiers after type reconstruction
- Type $T_4 \rightarrow T_4$ is a *type schema* that can be instantiated with any $T_4$ to make a type
- ML idea: let can bind identifiers to *polymorphic terms*
  - typing context maps variable to *type schema:* $\forall X_1,\ldots,X_n.\ \tau$ where $FTV(\tau) \subseteq \{X_1,\ldots,X_n\}$, $n \geq 0$
  - Type variables (parameters) $X$ can be replaced by any types if done consistently
  - e.g., let id = $\lambda$x.x gives id type schema $\forall X.X \rightarrow X$ so id can be used at types int→int, (bool→int)→(bool→int), …
- Can still do type inference! (ML)

# Typing let-polymorphism

$\Gamma ::= \varnothing \mid \Gamma, x{:}\sigma$

$\sigma ::= \tau \mid \forall X_1,\ldots,X_n.\ \tau$

$\Delta = X_1,\ldots,X_n$ ($\Delta$ : legal type variables)

$\Delta \vdash \tau$ (judgment: $\tau$ is well-formed)

$\underbrace{\Delta\ ;\ \Gamma}_{\text{Typing context}} \vdash e : \tau$ (judgment: e is well-formed)

$$\frac{\Delta\ ;\ \Gamma, x{:}\tau \vdash e : \tau' \qquad \Delta \vdash \tau \quad \Delta \vdash \tau'}{\Delta\ ;\ \Gamma \vdash \lambda x.e : \tau \to \tau'}$$

$$\frac{\Gamma(x) = \forall X_1, \ldots, X_n.\ \tau \qquad \Delta \vdash \tau_i \ \ \forall i \in 1..n}{\Delta\ ;\ \Gamma \vdash x : \tau\{\tau_i / X_i \ \ \forall i \in 1..n\}}$$



# More typing rules

Type parameters must be free

$$\frac{\Delta\ ;\ \Gamma \vdash e_1 : \tau \to \tau' \quad \Delta\ ;\ \Gamma \vdash e_2 : \tau \quad \Delta \vdash \tau, \tau'}{\Delta\ ;\ \Gamma \vdash e_1\ e_2 : \tau'}$$

$$\frac{\begin{array}{c}\Delta, X_1,\ldots,X_n \vdash \tau \quad \Delta \vdash \tau' \quad \Delta \cap \{X_1,\ldots,X_n\} = \varnothing \\ \Delta, X_1,\ldots, X_n\ ;\ \Gamma \vdash e_1 : \tau \quad \Delta\ ;\ \Gamma, x : \forall X_1,\ldots,X_n.\tau \vdash e_2 : \tau'\end{array}}{\Delta\ ;\ \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau'}$$

$$\frac{\dfrac{\{\alpha\};\ x{:}\alpha \vdash x : \alpha}{\{\alpha\}\ ;\ \varnothing \vdash (\lambda x.x) : \alpha \to \alpha} \qquad \dfrac{\varnothing;\ \text{id} : \forall\alpha.\alpha\to\alpha \vdash \text{id} : \text{int}\to\text{int}}{\varnothing;\ \text{id} : \forall\alpha.\alpha\to\alpha \vdash \text{id } 2 : \text{int}}}{\varnothing;\varnothing \vdash \text{let id} = (\lambda x.x) \text{ in id } 2 : \text{int}}$$

# Type well-formedness

- Well-formedness of types now more than conforming to a grammar
  - Can't mention unbound type variables
- Judgement $\Delta \vdash \tau$ means $\tau$ is well-formed in context $\Delta$

$$\frac{}{\Delta \vdash B} \qquad \frac{X \in \Delta}{\Delta \vdash X} \qquad \frac{\Delta \vdash \tau_1 \quad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 \to \tau_2}$$



# Algorithm $\mathcal{W}$ (Milner)

- Infers types in language with let-bound type schemas! (& letrec too)
- Strategy:
  - run type inference algorithm from last lecture to obtain type $\tau$ for each defined variable $x$
  - generalize *free type variables* $T_1 \ldots T_n$ of $\tau$ into type parameters $T_1, \ldots, T_n$
  - associate $x$ with type schema $\forall T_1, \ldots, T_n.\tau$
- **Generic**$(\tau, \Gamma, S) = \forall T_1, \ldots, T_n.S\tau$
  where $\{T_1, \ldots, T_n\} = FTV(S\tau) \setminus FTV(S\Gamma)$

# Inference algorithm

$\mathcal{W}(e, \Gamma, S) = \langle \tau, S' \rangle$ gives type, subst $S'$ as before, (but $\Gamma$ can map vars to type schemas)

Definition similar to $\mathcal{R}(e, \Gamma, S)$ but:

New type variables for every use!

$\mathcal{W}(x, \Gamma, S) = \text{case } \Gamma(x) \text{ of}$
$\quad | \ \forall T_1,\ldots,T_n.\tau \Rightarrow \langle \tau\{T_{fi}/T_i\}, S \rangle$

$\mathcal{W}(\text{let } x = e_1 \text{ in } e_2, \Gamma, S) =$
$\quad \text{let } \langle T_1, S_1 \rangle = \mathcal{W}(e_1, \Gamma, S) \text{ in}$
$\quad \text{let } \Gamma' = \Gamma[x \mapsto \textbf{Generic}(T_1, \Gamma, S_1)] \text{ in}$
$\quad\quad \mathcal{W}(e_2, \Gamma', S_1)$

$\mathcal{W}(\text{letrec } x = e_1 \text{ in } e_2, \Gamma, S) =$
$\quad \text{let } \Gamma' = \Gamma[x \mapsto T_f] \text{ in let } \langle T_1, S_1 \rangle = \mathcal{W}(e_1, \Gamma', S) \text{ in}$
$\quad \text{let } S_2 = \textbf{Unify}(\{T_f = T_1\}, S_1) \text{ in}$
$\quad \text{let } \Gamma'' = \Gamma[x \mapsto \textbf{Generic}(T_1, \Gamma, S_2)] \text{ in}$
$\quad\quad \mathcal{W}(e_2, \Gamma'', S_2)$



# Principal types

- Algorithms $\mathcal{W}, \mathcal{R}$ compute *principal types* for their respective languages: unique (up to alpha-equivalence on type variables), most-general types
- Not all type systems have principal types; e.g., unannotated Java
- Important property for building type-checker: there is always a best answer

# More polymorphism

- Still can't get as general a type as we'd like.
- Type of identity function: $\forall X.X \rightarrow X$
- Type of function that accepts an identity-like function and returns another one: $(\forall X_1.X_1 \rightarrow X_1) \rightarrow (\forall X_2.X_2 \rightarrow X_2)$
- Let-polymorphism: polymorphism on the outside
- Can get more expressive power by allowing type schemas to be used more like types
  - but we lose ML type inference



# Parametric polymorphism

- Polymorphism: expression has multiple types
- Parametric polymorphism $(\forall X_1,\ldots,X_n.\tau)$
  - types appear as parameters
  - expression is written the same way for all types
  - polymorphic value is *instantiated* on some types
- Java, C: no parametric polymorphism
  - Can't write generic code that doesn't care what are the types of values it manipulates

    `void sort(T[ ] arr)` —must say what T is!
    `Map m;` —can't define key, value type of m
    `v = m.get(k);` — no useful type checking
- C++, Modula-3: generic code through *templates*
  - (nearly) textual substitution

# Ad-hoc polymorphism

- Same name can be used to denote values with different types
- e.g. "+" refers to one operator on int, another on float, yet another on String
- Examples: C++, Java overloading
- Can be modeled by allowing overloading in the type context $\Gamma$
- Not true polymorphism: no polymorphic values



# Predicative polymorphism

$$e ::= x \mid \lambda x.\, e \mid e_0\, e_1 \mid \Lambda X.e \mid e[\tau]$$
$$\mid \lambda x{:}\, \tau.\, e \mid \lambda x{:}\, \sigma.\, e$$

type abstraction

type application

functions that operate on ordinary values

$$\tau ::= B \mid \tau_1 \rightarrow \tau_2$$
$$\sigma ::= \tau \mid \forall X.\sigma \mid \sigma_1 \rightarrow \sigma_2$$

functions that operate on polymorphic values

type schema

# Operational semantics

$e ::= x \mid \lambda x.\, e \mid e_0\, e_1 \mid \Lambda X.e \mid e[\tau]$
$\quad \mid \lambda x{:}\, \sigma.\, e$

$\tau ::= X \mid B \mid \tau_1 \to \tau_2$

$\sigma ::= \tau \mid \forall X.\sigma \mid \sigma_1 \to \sigma_2$

- Term application redn: $(\lambda x{:}\sigma.e_1)\, e_2 \to e_1\{e_2/x\}$
- Type application redn: $(\Lambda X.e)\, [\tau] \rightarrow e\{\tau/X\}$

- No effect on terms ⇒ just type-level book-keeping without computational significance
- Predicative = can only apply type abstractions to τ's



# Expressive power

- Can give types to many terms used in earlier λ-calculus encodings, e.g.

TRUE = $\Lambda X.\, \lambda x{:}X.\, \lambda y{:}X.\, x \;:\; \forall X.\, X \to X \to X$

FALSE = $\Lambda X.\, \lambda x{:}X.\, \lambda y{:}X.\, y \;:\; \forall X.\, X \to X \to X$

IF = $\Lambda Y.\, \lambda g{:}\forall X.\, X \to X \to X\,.$
$\qquad \lambda c{:}Y.\, \lambda a{:}Y.\, (g[Y]\; c\; a)$

$\quad : \; \forall Y.\, (\forall X.\, X \to X \to X) \to Y \to Y \to Y$

# Static Semantics

$\tau ::= B \mid X \mid \tau_1 \to \tau_2$
$\sigma ::= \forall X.\sigma \mid \tau \mid \sigma_1 \to \sigma_2$
$e ::= \lambda x{:}\sigma.e \mid e_1\, e_2 \mid \Lambda X.e \mid e[\tau]$
$\Gamma ::= \emptyset \mid \Gamma, x{:}\sigma$
$\Delta ::= \emptyset \mid \Delta, X$

*Judgements*:
$\Delta\,;\Gamma \vdash e : \sigma$
$\Delta \vdash \sigma$

$$\frac{}{\Delta\,;\Gamma, x{:}\sigma \vdash x : \sigma}$$

$$\frac{\Delta\,;\Gamma, x{:}\sigma \vdash e : \sigma'}{\Delta\,;\Gamma \vdash (\lambda x{:}\sigma.e) : \sigma \to \sigma'}$$

$$\frac{\Delta\,;\Gamma \vdash e_1 : \sigma \to \sigma' \quad \Delta\,;\Gamma \vdash e_2 : \sigma}{\Delta\,;\Gamma \vdash e_1\, e_2 : \sigma'}$$

$$\frac{\Delta, X\,;\Gamma \vdash e : \sigma}{\Delta\,;\Gamma \vdash \Lambda X.e : \forall X.\sigma} \qquad \frac{\Delta\,;\Gamma \vdash e : \forall X.\sigma}{\Delta\,;\Gamma \vdash e[\tau] : \sigma\{\tau/X\}}$$



# Modeling predicative polymorphism

- Need universe $\mathcal{U}$ of all ordinary type interpretations:
  $\mathcal{D}_0 = \{Z, U\}$
  $\mathcal{D}_n = \{X \to Y \mid X, Y \in \mathcal{D}_{n-1}\} \cup \mathcal{D}_{n-1}$
  $\mathcal{U} = \bigcup_{n \in \omega} \mathcal{D}_n$
- Each element of $\mathcal{U}$ is meaning of some type
- Environment $\chi$ maps type variables to domains

$\mathcal{T}[\![\text{int}]\!]\chi = Z$, $\mathcal{T}[\![1]\!]\chi = U$
$\mathcal{T}[\![\tau_1 \to \tau_2]\!]\chi = \mathcal{T}[\![\tau_1]\!]\chi \to \mathcal{T}[\![\tau_2]\!]\chi$
$\mathcal{T}[\![\sigma_1 \to \sigma_2]\!]\chi = \mathcal{T}[\![\sigma_1]\!]\chi \to \mathcal{T}[\![\sigma_2]\!]\chi$
$\mathcal{T}[\![X]\!]\chi = \chi(X)$
$\mathcal{T}[\![\forall X.\sigma]\!]\chi = \Pi_{D \in \mathcal{U}}\, \mathcal{T}[\![\sigma]\!]\chi[X \mapsto D]$

dependent product: set of all functions mapping $D \in \mathcal{U}$ to $\mathcal{T}[\![\sigma]\!]\chi[X \mapsto D]$

# Interpreting terms

- Given type variable environment $\chi$, ordinary variable environment $\rho$, such that $\chi \models \Delta$ and $\rho \models \Gamma$,

$$\mathcal{C}[\![\Delta;\Gamma \vdash e : \sigma]\!]\chi\rho \in \mathcal{T}[\![\sigma]\!]\chi$$

$\mathcal{C}[\![\Delta;\Gamma \vdash x : \sigma]\!]\chi\rho = \rho(x)$

$\mathcal{C}[\![\Delta;\Gamma \vdash e_1\, e_2 : \sigma']\!]\chi\rho = (\mathcal{C}[\![\Delta;\Gamma \vdash e_1 : \sigma \to \sigma']\!]\chi\rho)$
$(\mathcal{C}[\![\Delta;\Gamma \vdash e_2 : \sigma]\!]\chi\rho)$

$\mathcal{C}[\![\Delta;\Gamma \vdash \lambda x{:}\sigma.e : \sigma \to \sigma']\!]\chi\rho = \lambda v \in \mathcal{T}[\![\sigma]\!]\chi.$
$\mathcal{C}[\![\Delta;\Gamma, x{:}\sigma \vdash e : \sigma']\!]\chi\rho[x \mapsto v])$

$\mathcal{C}[\![\Delta;\Gamma \vdash \Lambda X.e : \forall X.\sigma]\!]\chi\rho = \lambda D \in \mathcal{U}.\, \mathcal{C}[\![\Delta,X;\Gamma \vdash e : \sigma]\!]\chi[X \mapsto D]\rho$

$\mathcal{C}[\![\Delta;\Gamma \vdash e[\tau] : \sigma\{\tau/X\}]\!]\chi\rho = (\mathcal{C}[\![\Delta;\Gamma \vdash e : \forall X.\sigma]\!]\chi\rho)(\mathcal{T}[\![\tau]\!]\chi)$

Substitution lemma: $\mathcal{T}[\![\sigma\{\tau/X\}]\!]\chi = \mathcal{T}[\![\sigma]\!]\chi[X \mapsto \tau]$



# Self-application

- What about $\lambda$x.(x x)? Still can't give it a type
- Still have strong normalization, as suggested by denotational semantics (and provable using logical relations)
- Next: the polymorphic lambda calculus (System F)